CS410
Advanced Functional Programming

Conor McBride
Mathematically Structured Programming Group
Department of Computer and Information Sciences
University of Strathclyde

October 18, 2014

Chapter 1

Introduction

1.1 Language and Tools

For the most part, we'll be using the experimental language, Agda Norell [2008], which is a bit like Haskell (and implemented in Haskell), but has a more expressive type system and a rather fabulous environment for typed programming. Much of what we learn here can be ported back to Haskell with a bit of bodging and fudging (and perhaps some stylish twists), but it's the programming environment that makes it worth exploring the ideas in this class via Agda.

The bad news, for some of you at any rate, is that the Agda programming environment is tightly coupled to the Emacs editor. If you don't like Emacs, tough luck. You may have a job getting all this stuff to work on whatever machines you use outside the department, but the toolchain all works fine on departmental machines.

Teaching materials, exercise files, lecture scripts, and so on, will all pile up in the repository `https://github.com/pigworker/CS410-14`, so you'll need to get with the git programme. We'll fix it so you each have your own place to put your official branch of the repo where I can get at it. All work and feedback will be mediated via your git repository.

1.2 Lectures, Lab, Tutorials

Monday: Lecture 11am–12pm, JA326; Lab, 3–5pm LT1301

Tuesday: Tutorial, 4–5pm, AR401a

Friday: Lecture, 1–2pm, LT210

1.3 Twitter @CS410afp

This class has a twitter feed. Largely, this is so that I can post pictures of the whiteboard. I don't use it for essential communications about class business, so you need neither join twitter nor follow this user. You can access all the relevant stuff just by surfing into `http://twitter.com/CS410afp`. This user, unlike my personal account, will follow back all class members who follow it, unless you ask it not to.

1.4 Hoop Jumping

CS410 Advanced Functional Programming is a level 4 class worth 20 credits. It is assessed *entirely* by coursework. Departmental policy requires class convenors to avoid deadline collisions by polite negotiation, so I've agreed the following dates for handins, as visible on the 4th year noticeboard.

- Monday week 3
- Monday week 6
- Monday week 9
- Monday week 12
- Monday week 15
- final assignment, issued as soon as possible after fourth year project deadline, to be submitted as late as I consider practicable before the exam board

Marking will happen 'live' and one-to-one, in office hours on a sign-up basis.

1.5 Getting Agda Going on Departmental Machines

Step 1. Use Linux. Get yourself a shell. (It's going to be that sort of a deal, all the way along. Welcome back to the 1970s.)

Step 2 for *bash* users. Ensure that your `PATH` environment variable includes the directory where Haskell's `cabal` build manager puts executables. Under normal circumstances, this is readily achieved by ensuring that your `.profile` file contains the line:

```
export PATH=$HOME/.cabal/bin:$PATH
```

After you've edited `.profile`, grab a fresh shell window before continuing.

Step 2 for *tcsh* users. Ensure that your `path` environment variable includes the directory where Haskell's `cabal` build manager puts executables. Under normal circumstances, this is readily achieved by ensuring that your `.cshrc` file contains the line:

```
set path = ($home/.cabal/bin $path)
```

After you've edited `.cshrc`, grab a fresh shell window before continuing.

Step 3. Ensure that you are in sync with the Haskell package database by issuing the command:

```
cabal update
```

Some people found that this bombs out with a missing library. Asking `which cabal` revealed that they had a spurious `~/.cabal/bin/cabal` file which took precedence over the regular /usr/bin/cabal. Simply delete ~/.cabal/bin/cabal to fix this problem.

Step 4. Install Agda by issuing the command:

```
cabal install agda
```

Yes, that's a lower case 'a' in 'agda'. In some situations, it may not manage the full installation in one go, delivering an error message about which package or version it has failed to install. We've found that it's sometimes necessary to do `cabal install happy` separately, and to do `cabal install alex-3.0`, requesting a specific older version, as required by another package.

Step 5. Wait.

Step 6. Wait some more.

Step 7. Assuming all of that worked just fine, set up the Emacs interactive environment with the command:

```
agda-mode setup; agda-mode compile
```

Step 8. Get this repository. Navigate to where in your file system you want to keep it and do

```
git clone https://github.com/pigworker/CS410-14.git
```

Step 9. Navigate into the repo.

```
cd CS410-14
```

Step 10. Start an emacs session involving an Agda file, e.g., by the command:

```
emacs Hello.agda &
```

The file should appear highlighted, and the mode line should say that the buffer is in Agda mode. In at least one case, this has proven problematic. To check what is going on, load the configuration file ~/.emacs and find the LISP command which refers to `agda-mode locate`. Try executing that command: select it with the mouse, then type ESC x, which should get you a prompt at which you can type `eval-region`, which will execute the selected command. If you get a message about not being able to find `agda-mode`, then edit the LISP command to give `agda-mode` the full path returned by asking `which agda-mode` in a shell. And if you get a bad response to `which agda-mode`, go back to step 2.

Step 11. When you're done, please confirm by posting a message on the class discussion forum.

1.6 Making These Notes

The sources for these notes are included in the repo along with everything else. They're built using the excellent `lhs2TeX` tool, developed by Andres Löh and Ralf Hinze. This, also, can be summoned via the Haskell package manager.

```
cabal install lhs2tex
```

With that done, the default action of `make` is to build these notes as `CS410.pdf`.

1.7 What's in `Hello.agda`?

It starts with a `module` declaration, which should and does match the filename.

```
module Hello where
```

Then, as in Haskell, we have comments-to-end-of-line, as signalled by `--` with a space.

```
-- Oh, you made it! Well done! This line is a comment.

-- In the beginning, Agda knows nothing, but we can teach it about numbers.
```

Indeed, this module has not `import`ed any others, and unlike in Haskell, there is no implicit 'Prelude', so at this stage, the only thing we have is the notion of a `Set`. The following `data` declaration creates three new things—a new `Set`, populated with just the values generated by its constructors.

```
data Nat : Set where
  zero  :  Nat
  suc   :  Nat -> Nat
```

We see some key differences with Haskell. Firstly, *one* colon means 'has type', rather than 'list cons'. Secondly, rather than writing 'templates' for data, we just state directly the types of the constructors. Thirdly, there's a lot of space: Agda has very simple rules for splitting text into tokens, so space is often necessary, e.g., around `:` or `->`. It is my habit to use even more space than is necessary for disambiguation, because I like to keep things in alignment.

Speaking of alignment, we do have the similarity with Haskell that indentation after `where` indicates subordination, showing that the declarations of the `zero` and `suc` value constructors belong to the declaration of the `Nat` type constructor.

Another difference is that I have chosen to begin the names of `zero` and `suc` in *lower* case. Agda enforces no typographical convention to distinguish constructors from other things, so we can choose whatever names we like. It is conventional in Agda to name data-like things in lower case and type-like things in upper case. Crucially, `zero`, `suc`, `Nat` and `Set` all live in the *same* namespace. The distinction between different kinds of things is achieved by referring back to their declaration, which is the basis for the colour scheme in the emacs interface.

The declaration of `Nat` tells us exactly which values the new set has. When we declare a function, we create new *expressions* in a type, but *no new values*. Rather, we explain which value should be returned for every possible combination of inputs.

```
-- Now we can say how to add numbers.

_+_ : Nat -> Nat -> Nat
zero   +  n  = n
suc m  +  n  = suc (m + n)
```

What's in a name? When a name includes *underscores*, they stand for places you can put arguments in an application. The unspaced _+_ is the name of the function, and can be used as an ordinary identifier in prefix notation, e.g. `_+_ m n` for `m + n`. When we use + as an infix operator (with arguments in the places suggested by the underscores), the spaces around it are necessary. If we wrote `m+n` by accident, we would find that it is treated as a whole other symbol.

Meanwhile, because there are no values in `Nat` other than those built by `zero` and `suc`, we can be sure that the definition of + covers all the possibilities for the inputs. Moreover, or rather, lessunder, the recursive call in the `suc` case has as its first argument a smaller number than in the pattern on the left hand side, so the recursive call is strictly simpler. Assuming (rightly, in Agda), that *values* are not recursive structures, we must eventually reach `zero`, so that every addition of values is bound to yield a value.

```
 -- Now we can try adding some numbers.

 four : Nat
 four = (suc (suc zero)) + (suc (suc zero))

 -- To make it go, select "Evaluate term to normal form" from the
 -- Agda menu, then type "four", without the quotes, and press return.

 -- Hopefully, you should get a response
 --   suc (suc (suc (suc zero)))
```

Evaluation shows us that although we have enriched our expression language with things like $2 + 2$, the values in `Nat` are exactly what we said they were: there are no new numbers, no error cases, no 'undefined's, no recursive black holes, just the values we declared.

That is to say, Agda is a language of *total* programs. You can approach it on the basis that things mean what they say, and—unusually for programming languages—you will usually be right.

1.8 Where are we going?

Agda is a language honest, expressive and precise. We shall use it to explore and model fundamental concepts in computation, working from concrete examples to the general structures that show up time and time again. We'll look at examples like parsers, interpreters, editors, and servers. We'll implement algorithms like arithmetic, sorting, search and unification. We'll see structures like monoids, functors, algebras and monads. The purpose is not just to teach a new language for instructing computers to do things, but to equip you with a deeper perception of structure and the articulacy to exploit that structure.

Agda is a dependently typed language, meaning that types can mention values and thus describe their intended properties directly. If we are to be honest and ensure that we mean what we say, we had better be able to say more precisely what we do mean. This is not intended to be a course in dependently typed programming, although precision is habit-forming, so a certain amount of the serious business is inevitable. We'll also be in a position to state and prove that the programs we write are in various ways sensible. What would it take to convince you that the + operator we constructed above really does addition?

I'm using Agda rather than Haskell for four reasons, two selfish, two less so.

- I am curious to see what happens.
- Using Agda brings my teaching a lot closer to my research and obliges me to generate introductory material which will help make this area more accessible. (The benefit for you is that I have lots of motivation to write thorough notes.)
- Agda's honesty will help us see things as they really are: we cannot push trouble under the rug without saying what sort of rug it is. Other languages are much more casual about run time failure or other forms of external interaction.
- Agda's editing environment gives strong and useful feedback during the programming process, encouraging a type-centred method of development, hopefully providing the cues to build good mental models of data and computation. We do write programs with computers: we don't just type them in.

Chapter 2

A Basic Prelude

Let us build some basic types and equipment for general use. We might need to rethink some of this stuff later, but it's better to keep things simple until life forces complexity upon us. In the course of establishing this setup, we'll surely encounter language features in need of explanation.

Concretely, we shall implement

```
module BasicPrelude where
```

the source code for this chapter of the notes is indeed that very module. We'll be able to import this module into others that we define later. The first exercise will put it to good use.

2.1 Natural Numbers

We have already had a quick preview of the datatype of natural numbers. Let us have it in our prelude.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
 {-# BUILTIN NATURAL Nat #-}
```

The funny comment-like BUILTIN things are not comments, but *pragmas*—not quite official parts of the language. Agda's implementers expected that we might need to define numbers, so these pragmas just tell Agda what we've chosen to call the bits and pieces. The payoff is that we are now allowed write numbers in *decimal*, leaving Agda to do all that succing.

If we define addition,

```
_+_ : Nat → Nat → Nat
zero   + n = n
suc m + n = suc (m + n)
infixr 5 _+_
```

then we can try evaluating expressions (using [C-c C-n]) such as

```
1 + 2 + 3 + 4
```

Note that the **infixr** *5* declaration assigns a precedence level of 5 to + (with higher binding more tightly) and ensures that multiple additions group to the right. The above means

```
1 + (2 + (3 + 4))
```

2.2 Impossible, Trivial, and Different

In this section, we build three finite types, capturing important basic concepts.

2.2.1 Zero

The Zero type has nothing after its **where** but silence. There is no way to make a *value* of type Zero. In Haskell, you could just write an infinite recursion or take the head of an empty list, but Agda won't countenance such dodges.

```
data Zero : Set where
```

The Zero type represents the idea of *impossibility*, which is a very useful idea, because if it's impossible to get into a situation, you don't need to worry about how to get out of it. The following definition bottles that intuition.

```
magic : {X : Set} →
        Zero → X
magic ()
```

There's plenty to explain here. The {*X* : Set} → means 'for all Sets, *X*'. So, the whole type says 'for all Sets *X*, there is a function from Zero to *X*'. To define a function, we must explain which value of the output type to return for each value of the input type. But that ought to be very easy, because there are no values of the input type! It's a bit like saying 'if you believe *that*, you'll believe anything'.

The braces have a secondary meaning: they tell Agda that we don't want to write *X* explicitly when we use magic. Rather, we want Agda to infer which *X* is relevant from the context in which magic is being used, just the same way that Haskell silently infers the types at which polymorphic functions are used. So, the first visible argument of magic has type Zero. If we're refining a goal by magic ?, then it's clear that *X* should be the goal type, and then we are left finding something of type Zero to fill in for the ?.

But how do way say what magic does? We don't. Instead, we say that it doesn't. The definition of magic is not given by an equation, but rather by *refutation*. In Agda, if we can point out that an input to a function is impossible, we do not have to write an = sign and an output. The way we point it out is to write the *absurd pattern* () in the place of the impossible thing. We're effectively saying 'BUSTED!'.

Note, by the way, that Agda's notation thus makes () mean the opposite of what it means in Haskell, where it's the empty tuple, easily constructed but not very informative. That's also a useful thing to have around.

2.2.2 One

tl;dr There is a type called One. It has one element, written ⟨⟩.

I could define a **data**type with one constructor. Instead, let me show you another feature of Agda—**record**s. Where a **data**type is given by a *choice* of constructors, a **record** type is given by a *collection* of fields. To build a record value, one must supply a value for each field. I define One to be the record type with *no fields*, so it is very easy to give a value for each field: there's only one way to do it.

```
record One : Set where
```

Values of **record** types are officially written **record** { *field1* = *value1* ; ... ; *fieldn* = *valuen* }, so the only value in One is

```
record { }
```

which is a bit of a mouthful for something so trivial. Fortunately, Agda lets us give a neater notation. We may optionally equip a record type with a *constructor*—the function which makes a record, taking the values of the fields as arguments. As part of the record declaration, I write

```
constructor ⟨⟩
```

which means, because there are no arguments, that

```
⟨⟩ : One
```

We are allowed to use either the official **record** notation or the constructor shorthand when we write patterns. Note that pattern matching an element of One does not tell us anything we didn't already know. Think:

Zero	impossible to make	useful to possess	not representable with bits
One	trivial to make	useless to possess	representable with no bits

On reflection, it is perhaps perverse to introduce record types with such a degenerate example. We'll have some proper records with fields in, shortly.

But first, let us complete our trinity of finite types by getting our hands on a bit, at last.

2.3 Two

The type Two represents a choice between exactly two things. As it is a choice, let's define it as a **data**type. As the two constructors have the same type, I can save space and declare them on the same line.

```
data Two : Set where
  tt ff : Two
```

In Haskell, this type is called Bool and has values True and False. I call the type Two to remind you how big it is, and I use ancient abbreviations for the constructors.

Agda's cunning mixfix syntax lets you rebuild familiar notations.

```
if_then_else_ : {X : Set} → Two → X → X → X
if tt then t else f  =  t
if ff then t else f  =  f
```

Again, we expect Agda to figure out the type of the conditional expression from the context, so we use braces to indicate that it should be hidden.

(Here are some dangling questions. Is it good that the types of the two branches are just the same as the type of the overall expression? Do we not know more, once we have checked the condition? How could we know that we know more?)

We can use conditionals to define conjunction of two Booleans:

```
_/\_ : Two → Two → Two
b1 /\ b2  =  if b1 then b2 else ff
```

Now that we have a way to represent Boolean values and conditional expressions, we might like to have some conditions. E.g., let us be able to compare numbers.

```
_≤_ : Nat → Nat → Two
zero  ≤ y       =  tt
suc x ≤ zero    =  ff
suc x ≤ suc y   =  x ≤ y
```

2.4 Lists

We can declare a **data**type which does the job of Haskell's workhorse $[a]$. The definition of List is parametrized by some X, the set in to which the list's elements belong. I write the parameters for the datatype to the left of the : in the declaration.

A typesetting gremlin prevents me from colouring [] red.

```
data List (X : Set) : Set where
  []   : List X
  _:>_ : X → List X → List X
infixr 5 _:>_
```

I give a 'nil' constructor, [], and a right associative infix 'cons' constructor, :>, which is arrowhead-shaped to remind you that you access list elements from left to right. We can write lists like

```
1 :> 2 :> 3 :> 4 :> 5 :> []
```

but Agda does not supply any fancy syntax like Haskell's $[1, 2, 3, 4, 5]$.

How many values are there in the set List Zero?

Does the set List One remind you of any other set that you know?

```
infixr 5 _++_
_++_ : {A : Set} → List A → List A → List A
[]          ++ ys  =  ys
(x :> xs)   ++ ys  =  x :> (xs ++ ys)
```

2.5 Interlude: Insertion

We've got quite a bit of kit now. Let's take a break from grinding out library components and write a program or two. In particular, as we have numbers and lists and comparison, we could write insertion sort. Let's see if we can remember how it goes. Split into cases, and the empty case is clear.

```
insertionSort : List Nat → List Nat
insertionSort []          = []
insertionSort (x :> xs)   = ?
```

What happens next? If we can insert x into the right place after sorting xs, we'll be home. Agda is a declare-before-use language, but a declaration does not have to be right next to the corresponding definition. We can make progress like this.

```
insertionSort : List Nat → List Nat
insertList : Nat → List Nat → List Nat
insertionSort []          = []
insertionSort (x :> xs)   = insertList x (insertionSort xs)


insertList y xs  = ?
```

Now, how do we insert? Again, we need to split the list into its cases. the [ ] case is easy. (It's also easy to get wrong.)

```
insertList y []          = y :> []
insertList y (x :> xs)   = ?
```

To proceed in the 'cons' case, we need to know whether or not y should come before x. We could go with

```
if y ≤ x then ? else ?
```

but let me take the chance to show you another feature. Instead of moving to the right and giving an expression, Agda lets us bring the extra information we need to the *left*, where we can pattern match on it.

```
insertList y []          = y :> []
insertList y (x :> xs) with y ≤ x
insertList y (x :> xs) |    b  = ?
```

The **with** construct adds an extra column to the left-hand side, tabulating cases for the result of the given expression. Now, if we split on b, we get

```
insertList y []          = y :> []
insertList y (x :> xs) with y ≤ x
insertList y (x :> xs) |    tt  = ?
insertList y (x :> xs) |    ff  = ?
```

and for each line of this extended table, it is clear what the output must be.

```
insertList y []          = y :> []
insertList y (x :> xs) with y ≤ x
insertList y (x :> xs) |    tt  = y :> x :> xs
insertList y (x :> xs) |    ff  = x :> insertList y xs
```

If the patterns to the left of the bar stay just the same as on the **with**-line, we're allowed to abbreviate them, as folllows.

```
insertList y []          =  y :> []
insertList y (x :> xs) with y ≤ x
...                    |  tt  =  y :> x :> xs
...                    |  ff  =  x :> insertList y xs
```

Which of these strikes you as a better document is a matter of taste.

2.5.1 Programs as Decision Trees

It's good to think of a function definition as the description of a *decision tree*. We start by considering a bunch of inputs and we need a strategy to deliver an output. We can

- give an output built from the stuff we've got, with the = *output* strategy;
- split one of our things into constructor cases, in each case considering the structures inside (and if there are no cases, we document that with an absurd pattern);
- get more stuff to consider by asking the value of some *extra* expressed in terms of the stuff we already have—that's what the right-hand side, **with** *extra*, achieves.

You can read a program as a dialogue between the machine, saying 'what am I supposed to do with this stuff?' on the left, and the programmer, explaining how to proceed on the right by one of the above strategies. The case-splitting nodes aren't documented by an explicit right-hand-side in the final program, but you see them in passing while you work, and you can see that they result in multiple left-hand sides for distinguished cases. Agda figures out how to compute your functions by reconstructing the full decision tree from the constructors in your patterns.

2.6 Unit Testing with Dependent Types

I'm only in chapter two and I can't resist the temptation. I want to be able to write unit tests in my code—example expressions which should have the values given, e.g.

```
insertionSort (5 :> 2 :> 4 :> 3 :> 1 :> []) == (1 :> 2 :> 3 :> 4 :> 5 :> [])
```

The good news is that Agda can run old programs *during* typechecking of new programs. We can make the typechecker run our unit tests for us, making use of the following piece of kit.

Yes, it is scary.

```
infix 4 _==_
data _==_ {X : Set} (x : X) : X → Set where
  refl : x == x
```

This **data**type has two parameters: the X in braces is a Set, and the braces, as ever, mean that it should be hidden; the x is an element of X, and its *round* brackets mean that it should be *visible*, in this case as the thing to the left of the == sign. However, right of the :, we have X → Set, not just Set, because this is an *indexed* collection of sets. For each y : X, we get a set x == y whose elements represent *evidence* that x and y are equal. The constructor tells us the only way to generate the evidence. The return type of a constructor may choose any value for the index, and it delivers values only for that index. Here, by choosing x for the index in the type of refl, we ensure that for equality evidence to exist, the two sides of the equation must have the very same value.

The upshot of all this is that we can write a unit test like this:

```
iTest : insertionSort (5 :> 2 :> 4 :> 3 :> 1 :> []) == (1 :> 2 :> 3 :> 4 :> 5 :> [])
iTest = refl
```

The typechecker must make sure it is valid to use the refl constructor, so it evaluates both sides of the equation to ensure that they are the same.

Try messing up the program to see what happens!

Even better, try deleting the program and rebuilding it interactively. While your program is under construction and the test might possibly work out fine in the end, the refl evidence in the unit test will have a yellow background, indicating that it is suspicious . But you will not be allowed to do anything interactively which makes the test actually fail, and if you override the interactive system and load a silly program, the refl will have the brown background of steaming unpleasantness .

sus-pish-ous

We won't be fooling around with fancy types in programming for a while yet, but unit testing is a good engineering practice, so let us take advantage of Agda's capacity to support it.

2.7 More Prelude: Sums and Products

We often build datatypes which offer some sort of choice. Sometimes we just want to give a choice between two types which are already established. The type which offers 'an S or a T' is called the *sum* of S and T. We define it as a datatype with S and T as parameters, allowing constructors, for 'left injection' and 'right injection', respectively.

Haskell calls this construction *Either*.

```
infixr 1 _/+/_
data _/+/_ (S T : Set) : Set where
  inl : S → S /+/ T
  inr : T → S /+/ T
```

To see why it really is a kind of sum, try finding all the *values* in each of

```
Zero /+/ Zero   Zero /+/ One   One /+/ One   One /+/ Two   Two /+/ Two
```

When we offer a choice, we need to able to cope with either possibility. The following gadget captures the idea of 'computing by cases'.

```
_⟨?⟩_ : {S T X : Set} →
        (S → X) → (T → X) →
        S /+/ T → X
```

```
(f ⟨?⟩ g) (inl s)  =  f s
(f ⟨?⟩ g) (inr t)  =  g t
```

It might look a bit weird that it's an *infix* operator with *three* arguments, but it's used in a higher-order way. To make a function, $f\ \langle ? \rangle\ g$ which takes $S\ /{+}/\ T$ to some X, you need to have a function for each case, so f in $S \rightarrow X$ and g in $T \rightarrow X$.

Meanwhile, another recurrent theme in type design is that we ask for a *pair* of things, drawn from existing types. This is, somehow, the classic example of a **record**.

Haskell uses the notation $(s{,}t)$ for both the types and values.

```
infixr 2 _/×/_
record _/×/_ (S T : Set) : Set where
  constructor _,_
  field
    outl : S
    outr : T
open _/×/_ public
infixr 4 _,_
```

I have a little explaining to do, here. The **field** keyword introduces the declarations of the record's fields, which must be indented below it. We have two fields, so it makes sense to have an infix **constructor**, which is just a comma—unlike Haskell, parentheses are needed only to resolve ambiguity. The **open** declaration makes outl and outr available as the 'left projection' and 'right projection' functions, respectively. You can check that

```
outl : {S T : Set} → S /×/ T → S      outr : {S T : Set} → S /×/ T → T
```

The **public** means that outl and outr stay in scope whenever any other module imports this one.

To see why $S\ /{\times}/\ T$ is called the *product* of S and T, try finding all the values in the following types.

```
Zero /×/ Zero   Zero /×/ One   One /×/ One   One /×/ Two   Two /×/ Two
```

It is sometimes useful to be able to convert a function which takes a pair into a function which takes its arguments one at a time. This conversion is called 'currying' after the logician, Haskell Curry, even though Moses Schon̈finkel invented it slightly earlier.

```
curry : {S T X : Set} →
        (S /×/ T → X) →
        S → T → X
curry f s t = f (s,t)
```

Its inverse is, arguably, even more useful, as it tells you how to build a function from pairs by considering each component separately.

```
uncurry : {S T X : Set} →
          (S → T → X) →
          S /×/ T → X
uncurry f (s,t) = f s t
```

2.8 Interlude: Exponentiation

How many functions are there in a type $S \to T$? It depends on when we consider two functions to be the same. Mathematically, such a function is considered just to be the choice of a T value corresponding to each S value. There might be lots of different ways to *implement* that function, but if two programs of type $S \to T$ agree on outputs whenever we feed them the same inputs, we say they are two implementations of the same function.

So it's easy to count functions, at least if the sets involved are finite. If there are t different elements of T and s different elements of S, then we need to choose one of the t for each one of the s, so that's t^s different possibilities. Just as $S\ /{+}/\ T$ really behaves like a sum and $S\ /{\times}/\ T$ really behaves like a product, we find that $S \to T$ really behaves like the exponential T^S.

The fact that curry and uncurry are mutually inverse (or *isomorphic*) just tells us something we learned in school

$$X^{(S\ /{\times}/\ T)} \cong (X^T)^S$$

You might also remember that

$$X^{(S\ /{+}/\ T)} \cong X^S /{\times}/ X^T$$

and it's not hard to see why that makes sense in terms of counting functions. (Think about what $\langle ? \rangle$ does.)

Many of the algebraic laws you learned for numeric operations at school make perfect sense for *type* operations and account for structures fundamental to computation. That's (to some extent) how the Mathematically Structured Programming Group came by its name. Keep your eyes peeled for more!

2.9 More Prelude: Basic Functional Plumbing

Functions are a bit like machines with an input pipe and an output pipe. Their types tell us whether it's safe to plumb them together. Any functional plumber needs some basic tools.

Firstly, here's a bit of pipe with no machine in the middle—the *identity* function. What comes out is what went in!

```
id : {X : Set} → X → X
id x = x
```

Secondly, we need to be able to plumb the output from one machine to the input of another. Here's function *composition*.

```
_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
(f ∘ g) a = f (g a)
infixr 2 _∘_
```

What laws do you think id and ∘ should obey? If you plumb an extra bit of pipe onto a machine, does it change what the machine does? If you plumb a sequence of machines together, the order of the machines can clearly matter, but does the order in which you did the plumbing jobs affect the behaviour of the end product?

Chapter 3

Logic via Types

The inescapable honesty of Agda makes it possible for us to treat values as *evidence* for something. We gain a logical interpretation of types.

```
module Logic where
open import BasicPrelude
```

One way of looking at logical formulae is to consider what constitutes evidence that they hold. We can look at the connectives systematically.

What constitutes 'evidence for A or B'? Either 'evidence for A' or 'evidence for B'. If we have a type, A, representing 'evidence for A' and another, B representing 'evidence for B', then A /+/ B represents 'evidence for A or B'.

What constitutes 'evidence for A and B'? We need both 'evidence for A' and 'evidence for B'. If we have a type, A, representing 'evidence for A' and another, B representing 'evidence for B', then A /×/ B represents 'evidence for A and B'.

What constitutes 'evidence that A implies B'? We need to be sure that, given 'evidence for A', we can produce 'evidence for B'. If we have a type, A, representing 'evidence for A' and another, B representing 'evidence for B', then $A \to B$ represents 'evidence for A and B'.

There will be more to say here, after exercise 1 is completed, but the basic message is:

propositions are types; types are propositions
proofs are programs; programs are proofs

Types like Nat are rather boring propositions. Types like $2 + 2 == 4$ are slightly more interesting.

Chapter 4

Hutton's Razor

This chapter is inspired by Professor Graham Hutton, author of *Programming in Haskell*. We investigate various topics in the semantics (static and dynamic) of programming languages by considering minimal extensions to a very simple programming language—the expressions built from addition and natural numbers. The idea is that adding up numbers is indicative of 'ordinary computation' in general, and we need add only the extra features required to expose whatever departure from the ordinary we care about. Why use complicated examples when simple ones will do? Graham champions simplicity, and thus gives very clear explanations of important things. His friends call the adding-up-numbers language *Hutton's Razor* in his honour.

```
module Razor where
open import BasicPrelude
```

Without further ado, let us have a datatype of 'Hutton expressions'.

```
data HExp : Set where
  val   : Nat → HExp
  _⧺_  : HExp → HExp → HExp
```

Evaluating expressions is quite easy. Let us do it. The essence of it is to replace the type HExp of syntactic things with a type of semantic values, in this case, Nat itself. To do that, we need to replace the constructors, which make syntactic things, with semantic counterparts. In effect, val becomes id and ⧺ becomes +.

```
Val : Set
Val = Nat

eval : HExp → Val
eval (val n)       = n    -- which is equal to id n
eval (e₁ ⧺ e₂)  = eval e₁ + eval e₂
```

So

```
eval ((val 1 ⧺ val 2) ⧺ val 3) = ((id 1 + id 2) + id 3) = 6
```

We could think up other ways to interpret the syntax of Hutton's Razor. We might, for example, just collect the list of numerical constants which occur in an expression.

```
constants : HExp → List Nat
constants (val n)     = n :> []
constants (e1 +++ e2) = constants e1 ++ constants e2
```

An interpretation of a bunch of operators for a given 'value type' T is called an **algebra** for those operators: T is called the **carrier** of the algebra. That is id and $+$ give an algebra for HExp with carrier Val; $(\lambda\ n \rightarrow n$:> []) and ++ give an algebra for HExp with carrier List Nat. Of course, val and +++ give an algebra for HExp with carrier HExp, and we call it the **free** algebra because it is not obliged to obey any mathematical laws. E.g., we expect $+$ to be associative, and more to the point, we expect $2 + 2 = 4$, but val 2 +++ val 2 is *not* the same *expression* as val 4. So you can think of 'free' as meaning 'nothing more than syntax'.

For the rest of this chapter, we shall explore variations on the theme of Hutton's Razor, illustrative of all sorts of computational ideas, in search of common patterns.

4.1 Compiling for a Stack Machine

Our first variation is to consider *compiling* rather than *interpreting* Hutton's Razor. The compilation target will be a simple machine which manages the evaluation process via a *stack*. The machine has just two instructions:

PUSH n puts the number n on top of the stack;

ADD pops n and then m from the stack, then pushes $m + n$.

Our mission is to write a compiler which turns an expression into a sequence of these instructions which leaves the expression's value on the top of the stack, with the original stack below. For example, compiling (val 2 +++ val 3) +++ val 1 should give us

```
code                  stack after instruction

                      ...
PUSH 2                2 , ...
PUSH 3                3 , 2 , ...
ADD                   5 , ...
PUSH 1                1 , 5 , ...
ADD                   6 , ...
```

Correspondingly, let me give a type for code sequences. (I grew up in the 1970s, so assembly language operators are in ALL CAPS.)

```
data HCode : Set where
  PUSH    : Nat → HCode
  ADD     : HCode
  _-SEQ-_ : HCode → HCode → HCode
infixr 3 _-SEQ-_
```

That is, we have our two instructions, along with a *sequential composition* operator, allowing us to plug programs together. I could have used *lists* of instructions, but this gives a slightly simpler and more extensible treatment: watch this space.

Based on our informal description of what the stack machine code should do, we can have a go at implementing the compiler.

```
compile : HExp → HCode
compile (val n)       = PUSH n
compile (e₁ +++ e₂)   = compile e₁ −SEQ− compile e₂ −SEQ− ADD
```

That is, we have implemented yet another algebra for HExp, this time with carrier HCode, with PUSH for val and λ c_1 c_2 → c_1 −SEQ− c_2 −SEQ− ADD for +++.

However, if we want to see it working, we need to say how to *execute* programs in HCode. We need to write some

```
exec : HCode → List Nat → List Nat
```

interpreting code as a function from the initial stack (represented as a list) to the final stack. We need to give an algebra for HCode with carrier List Nat → List Nat. But when we try, we hit a snag.

```
exec : HCode → List Nat → List Nat
exec (PUSH x)      s               = x :> s
exec ADD           []              = { }0
exec ADD           (n :> [])       = { }1
exec ADD           (n :> (m :> s)) = (m + n) :> s
exec (c₁ −SEQ− c₂) s               = exec c₂ (exec c₁ s)
```

We are obliged to say what happens when there are not enough values in the stack for the correct execution of ADD. How will we handle stack *underflow*? We could just give back any old rubbish, but then we would have to worry that the old rubbish we give back might actually happen, when we intend that underflow should *never* happen. Moreover, if we have been competent, at least the HCode which comes from compile will never cause underflow.

It is clear that we need to manage the height of the stack somehow, and Agda lets us do it with *types*. We can express requirements and guarantees about the length of lists by working with the type of *vectors*, or 'lists indexed by their length'.

```
data Vec (X : Set) : Nat → Set where
  []   : Vec X zero
  _:>_ : {n : Nat} → X → Vec X n → Vec X (suc n)
```

The type constructor Vec now has the type

```
Vec : Set → Nat → Set
```

so that, for example, Vec Two 8 is the type of bit-vectors of length *exactly* 8. You can see that, as with list, we have a type (X : Set) of elements declared left of the : in the head of the **data** declaration, so it scopes over the whole of the rest of the declaration. By writing Nat → Set instead of Set to the right of :, we are saying that we want a whole *family* of vector types, one for each length, and that we want to be free to choose the lengths that show up in the constructor types. Specifically, we fix it so that [] makes only vectors with length zero and that x :> xs has length one more than xs. The curly braces in {n : Nat} → mean that the n which gives the length of the vector's tail will be kept hidden by default.

By the same facility, we can refine HCode with our knowledge of how instructions affect stack height. This time, we index by two numbers, respectively representing the initial and final heights of the stack when the code is executed.

```
data HCode : Nat → Nat → Set where
  PUSH    : {i : Nat} → Nat → HCode i (suc i)        -- one goes on
  ADD     : {i : Nat} → HCode (suc (suc i)) (suc i)  -- two off, one on
  _-SEQ-_ : {i j k : Nat} → HCode i j → HCode j k → HCode i k
                                             -- dominoes!
```

We can use ADD only in situations where it is known that there is enough stuff on the stack. We have made underflow *impossible*!

Now, we can give exec a type that makes the promise implicit in our claim that the indices of HCode represent the initial and final stack heights: we should input a stack of the initial height and output a stack of the final height. The code keeps the promise.

```
exec : {i j : Nat} → HCode i j → Vec Nat i → Vec Nat j
exec (PUSH x)        s              = x :> s
exec ADD             (x :> (y :> s)) = (y + x) :> s
exec (c1 -SEQ- c2) s               = exec c2 (exec c1 s)
```

If you build this program, you will see that the underflow cases do not even appear. To see what is going on, let me use the 'manual override' syntax to show you some of the hidden numbers.

```
exec : {i j : Nat} → HCode i j → Vec Nat i → Vec Nat j
exec .{i}             .{suc i} (PUSH {i} n)                  s
  = n :> s
exec .{suc (suc i)} .{suc i} (ADD {i})                     (n :> (m :> s))
  = (m + n) :> s
exec .{i}             .{k}     (_-SEQ-_ {i} {j} {k} c1 c2) s
  = exec {j} {k} c2 (exec {i} {j} c1 s)
```

The curly braces show the height information which is usually hidden in the calls to exec and in the HCode *I* constructors. Some of the patterns are marked with a .which you can read as 'no choice but'. When splitting gives us the case that the code was PUSH {i} n, the initial stack height must be i and the final stack height must be suc i, because nothing else fits the types we have given: fortunately, n :> s is a vector one longer than s. Similarly, in the ADD case, the initial height must be at least two, so when we match on the stack, the underflow cases do not fit the known information and are rejected by Agda as impossible.

When we split a scrutinee, for each constructor case Agda learns that the indices of the scrutinee's type must equal those of the constructor's return type, and she tries to solve the equations: sometimes she learns that the case cannot happen; sometimes she learns useful information about the indices. Crucially, as soon as we have types which are related to each other by indexing, inspecting data (e.g., the HCode code) no longer happens in isolation. Instead, we can learn useful information about other data (e.g., the stack).

To finish the job, we need to fix up compile. Here, we must be sure that the code we generate can run starting from any height of stack. Further, we can promise that the stack will only grow by one: that promise used to be *waffle*, but now it is in the type and checked! Happily, our old compile function really did keep the promise, so it typechecks without further ado.

```
compile : HExp → {i : Nat} → HCode i (suc i)
compile (val n)     = PUSH n
compile (e1 +++ e2) = compile e1 -SEQ- compile e2 -SEQ- ADD
```

We have used types to eliminate underflow and to keep the promise that compiled expressions grow the stack by one value. However, we might like to know that the compiler really works. It is not enough just to push some value: we need to know that we get the value we expect—the value given by eval. Our == type allows us to express that very property, and we may then write a *program* which computes the evidence for that property.

```
correct : (e : HExp) {i : Nat} (s : Vec Nat i) →
          exec (compile e) s == eval e :> s
correct (val n)      s = refl
correct (e1 +++ e2) s
  rewrite correct e1 s
     |    correct e2 (eval e1 :> s)
  = refl
```

It is a little difficult to do the process justice on paper. When the expression is val n, we discover that our return type is

```
exec (compile (val n)) s == eval (val n) :> s
```

but, just by the computation rules we wrote in our programs,

```
exec (compile (val n)) s = exec (PUSH n) s = n :> s
```

and

```
eval (val n) :> s = n :> s
```

so both sides of the equation are already equal, and the refl constructor typechecks!

We have a harder time of it with addition. After computation, the goal is

```
exec ADD (exec (compile e2) (exec (compile e1) s)) == (eval e1 + eval e2) :> s
```

and we can get no further. However, we could make a recursive call on a subexpression,

```
correct e1 s : exec (compile e1) s == eval e1 :> s
```

If you have learned *proof by induction*, you may have encountered the notion of 'inductive hypothesis'. For us that is exactly 'the type of a permitted recursive call'. The special **rewrite** syntax works specifically with the == type, allowing us to rewrite the goal by any equation we can prove. After the first step, we get

```
correct : (e : HExp) {i : Nat} (s : Vec Nat i) →
          exec (compile e) s == eval e :> s
correct (val n)      s = refl
correct (e1 +++ e2) s
  rewrite correct e1 s
  = { }0
```

where we have made some definite progress.

```
{ }0 : exec ADD (exec (compile e2) (eval e1 :> s)) == (eval e1 + eval e2) :> s
```

Agda allows us a further rewrite by glueing on another proof with the | symbol. This time, we can use the inductive hypothesis for the other subexpression, giving the stack that the goal presents us,

```
correct e2 (eval e1 :> s) : exec (compile e2) (eval e1 :> s) == eval e2 :> eval e1 :> s
```

so that the goal becomes

```
exec ADD (eval e₂ :> eval e₁ :> s) == (eval e₁ + eval e₂) :> s
```

Now,xs the left-hand side computes to the right-hand side, so refl brings us home.

You can compute

```
exec (compile (val 2 +H+ val 2))
```

and be glad of the answer 4, but be aware that you now work in a world where you can show that your program is *always* correct, for all possible inputs.

4.2 Hutton's Razor with Variables

So far, we have considered *closed* expressions built from operators and constants. When we implement functions, we often work with *open* expressions which also contain *variables*, e.g., the formal parameters of the functions. We can extend our little syntax to allow for this possibility.

```
data HExp (V : Set) : Set where
  var    : V → HExp V
  val    : Nat → HExp V
  _+H+_  : HExp V → HExp V → HExp V
```

Our type has acquired a parameter V, being the set of variables which may occur in expressions, and an extra constructor, var which makes every variable an expression. I encourage you to think of V not as the set of syntactically valid identifiers, but rather as the set of variables *in scope*. Correspondingly, we should be able to evaluate expressions if we give a value for each of the variables which may occur in them. An assignment of values to variables is called an **environment**, and here we may represent such a thing simply as a function of type $V \to$ Val.

```
eval : {V : Set} → HExp V → (V → Val) → Val
eval (var x)       γ = γ x
eval (val n)       γ = n
eval (e₁ +H+ e₂) γ = eval e₁ γ + eval e₂ γ
```

For example, we might have two variables, represented by the elements of Two.

```
myHExp : HExp Two
myHExp = ((val 1 +H+ var tt) +H+ (var tt +H+ var ff))
```

```
(eval myHExp λ {tt → 7; ff → 11}) == 26
```

Notation. In Agda, the scope of a λ extends as far as possible to the right, so there is a common tendency not to parenthesize λ-abstractions when they are the last argument to a function. Meanwhile, the braces-and-semicolons notation allows you to do a little bit of local case-splitting inside a λ-abstraction, used here to give a different value to each variable.

We can recover plain Hutton's Razor by making variables impossible: all expressions in HExp Zero are *closed*. Indeed one motivation for bothering with Zero is to give a uniform treatment to open and closed expressions, but recover closed expressions specifically whenever we need them. If we have no variables, we can evaluate without an environment as before, or rather, it is easy to come up with an environment which assigns values to all none of the variables.

```
eval0 : HExp Zero → Nat
eval0 e  =  eval e magic
```

4.3 Simultaneous Substitution for Open Expressions

Environments map variables to values, but that is not all we can do with variables in expressions. In particular, we may systematically replace variables by expressions, performing *substitution*.

```
subst : {U V : Set} → HExp U → (U → HExp V) → HExp V
subst (var x)       σ  =  σ x
subst (val n)       σ  =  val n
subst (e₁ ⧻ e₂) σ  =  subst e₁ σ ⧻ subst e₂ σ
```

The function $\sigma : U \to$ HExp V maps each variable in U to some expression whose variables are drawn from V. We can roll out this substitution to all the variables in a HExp U to get a HExp V: because we act on all the variables, not just one of them, we call this operation *simultaneous* substitution.

Substitution is a lot like evaluation, but where eval extends our original evaluation algebra with an environment, subst extends the original free algebra with the substitution. We can prove a useful fact relating the two.

```
evalSubstFact : {U V : Set} (e : HExp U) (σ : U → HExp V) (γ : V → Val) →
                (eval (subst e σ) γ) == (eval e λ u → eval (σ u) γ)
evalSubstFact (var x) σ γ  =  refl
evalSubstFact (val n) σ γ  =  refl
evalSubstFact (e₁ ⧻ e₂) σ γ
  rewrite evalSubstFact e₁ σ γ | evalSubstFact e₂ σ γ
  =  refl
```

If you substitute then evaluate, you get the same answer as if you build an environment from the substitution. The proof is an easy induction on expressions, rewriting the goal for e_1 ⧻ e_2 by both inductive hypotheses. Why is this fact useful? It tells us that the typical way we evaluate function calls is sensible: when we give the *actual* parameters to a function, they are expressions, and what we do in theory is substitute those *expressions* for the formal parameters and evaluate, but what we really do is evaluate the actual parameters to construct an environment which assigns *values* to the formal parameters.

A special case of this useful fact is that, if we had to, we could make do with evaluation for *closed* terms only by turning the environment into a substitution, replacing each variable by a numerical constant.

```
eval0Fact : {V : Set} (e : HExp V) (γ : V → Val) →
            eval0 (subst e (val ∘ γ)) == eval e γ
eval0Fact e γ  =  evalSubstFact e (val ∘ γ) magic
```

Appendix A

Agda Mode Cheat Sheet

I use standard emacs keystroke descriptions. E.g., ‘C-c’ means control-c. I delimit keystrokes with square brackets, but don’t type the brackets or the spaces between the individual key descriptions.

A.1 Managing the buffer

[C-c C-l] load buffer

This keystroke tells Agda to resynchronize with the buffer contents, typechecking everything. It will also make sure everything is displayed in the correct colour.

[C-c C-x C-d] deactivate goals

This keystroke deactivates Agda’s goal machinery.

[C-c C-x C-r] restart Agda

This keystroke restarts Agda.

A.2 Working in a goal

The following apply only when the cursor is sitting inside the braces of a goal.

[C-c C-,] what’s going on?

If you select a goal and type this keystroke, the information buffer will tell you the type of the goal and the types of everything in the context. Some things in the context are not in scope, because you haven’t bound them with a name anywhere.

These show up with names Agda chooses, beginning with a dot: you cannot refer to these things, but they do exist.

[C-c C-.] more on what's going on?

This is a variant of the above which in addition also shows you the type of the expression currently typed into the hole. This is useful for trying different constructions out before giving/refining them!

[C-c C-spc] give expression

If you think you know which expression belongs in a goal, type the expression between its braces, then use this keystroke. The expression can include ? symbols, which become subgoals.

[C-c C-c] case split

If your goal is immediately to the right of =, then you're still building your program's decision tree, so you can ask for a case analysis. Type the name of a variable in the goal, then make this keystroke. Agda will try to split that variable into its possible constructor patterns. Amusingly, if you type several variables names and ask for a case analysis, you will get all the possible combinations from splitting each of the variables.

[C-c C-r] refine

If there's only one constructor which fits in the hole, Agda deploys it. If there's a choice, Agda tells you the options.

[C-c C-a] ask Agsy (a.k.a. I feel lucky)

If you make this keystroke, Agda will use a search mechanism called 'Agsy' to try and guess something with the right type. Agsy may not succeed. Even if it does, the guess may not be the right answer. Sometimes, however, there's obviously only one sensible thing to do, and then Agsy is your bezzy mate! It can be an incentive to make your types precise!

A.3 Checking and Testing things

[C-c C-d] deduce type of expression

If you type this keystroke, you will be prompted for an expression. If the expression you supply makes sense, you will be told its type.

If you are working in a goal and have typed an expression already, Agda will assume that you want the type of that expression.

[C-c C-n] normalize expression

If you type this keystroke, you will be prompted for an expression. If the expression you supply makes sense, you will be told its value.

If you are working in a goal and have typed an expression already, Agda will assume that you want to normalize (i.e. compute as far as possible) that expression. The normal form might not be a value, because there might be some variables in your expression, getting in the way of computation. When there are no free variables present, the normal form is sure to be a value.

A.4 Moving around

[C-c C-f]/[C-c C-b] move to next/previous goal

A quick way to get to where the action is to use these two keystrokes, which takes you to the next and previous goal respectively.

[M-.] go to definition

If you find yourself wondering what the definition of some identifier is, then you can put the cursor at it and use this keystroke – it will make Agda take you there.

Bibliography

Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.